Introduce `IO::Buffer` for fiber scheduler.

07/03/2021 07:24 AM - ioquatix (Samuel Williams)

<table>
<thead>
<tr>
<th>Status:</th>
<th>Closed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority:</td>
<td>Normal</td>
</tr>
<tr>
<td>Assignee:</td>
<td></td>
</tr>
<tr>
<td>Target version:</td>
<td></td>
</tr>
</tbody>
</table>

**Description**

After continuing to build out the fiber scheduler interface and the specific hooks required for io_uring, I found some trouble within the implementation of IO.

I found that in some cases, we need to read into the internal IO buffers directly. I tried creating a "fake string" in order to transit back into the Ruby fiber scheduler interface and this did work to a certain extent, but I was told we cannot expose fake string to Ruby scheduler interface.

So, after this, and many other frustrations with using String as a IO buffer, I decided to implement a low level IO::Buffer based on my needs for high performance IO, and as part of the fiber scheduler interface.

Here is roughly the interface implemented by the scheduler w.r.t. the buffer:

```ruby
class Scheduler
  # @parameter buffer [IO::Buffer] Buffer for reading into.
  def io_read(io, buffer, length)
    # implementation provided by `read` system call, IO_URING_READV, etc.
  end

  # @parameter buffer [IO::Buffer] Buffer for writing from.
  def io_write(io, buffer, length)
    # implementation provided by `write` system call, IO_URING_WRITEV, etc.
  end

  # Potential new hooks (Socket#recvmsg, sendmsg, etc):
  def io_recvmsg(io, buffer, length)
  end
end
```

In reviewing other language designs, I found that this design is very similar to Crystal's IO buffering strategy.

The proposed implementation provides enough of an interface to implement both native schedulers as well as pure Ruby schedulers. It also provides some extra functionality for interpreting the data in the buffer. This is mostly for testing and experimentation, although it might make sense to expose this interface for binary protocols like HTTP/2, QUIC, WebSockets, etc.

**Proposed Solution**

We introduce new class IO::Buffer.

```ruby
class IO::Buffer
  # @returns [IO::Buffer] A buffer with the contents of the string data.
  def self.for(string)
  end

  PAGE_SIZE = # ... operating system page size

  # @returns [IO::Buffer] A buffer with the contents of the file mapped to memory.
  def self.map(file)
  end

  # Flags for buffer state.
  EXTERNAL = # The buffer is from external memory.
  INTERNAL = # The buffer is from internal memory (malloc).
  MAPPED = # The buffer is from mapped memory (mmap, VirtualAlloc, etc)
end
```

03/20/2022
# The buffer is locked for usage (cannot be resized)
PRIVATE = # The buffer is mapped as copy-on-write.
IMMUTABLE = # The buffer cannot be modified.

# @returns [IO::Buffer] A buffer with the specified size, allocated according to the given flags.
def initialize(size, flags)
  end

# @returns [Integral] The size of the buffer
attr :size

# @returns [String] A brief summary and hex dump of the buffer.
def inspect
  end

# @returns [String] A brief summary of the buffer.
def to_s
  end

# Flag predicates:
def external?
  end
def internal?
  end
def mapped?
  end
def locked?
  end
def immutable?
  end

# Flags for endian/byte order:
LITTLE_ENDIAN = # ...
BIG_ENDIAN = # ...
HOST_ENDIAN = # ...
NETWORK_ENDIAN= # ...

# Lock the buffer (prevent resize, unmap, changes to base and size).
def lock
  raise "Already locked!" if flags & LOCKED
  flags |= LOCKED
end

# Unlock the buffer.
def unlock
  raise "Not locked!" unless flags & LOCKED
  flags |= ~LOCKED
end

// Manipulation:
# @returns [IO::Buffer] A slice of the buffer's data. Does not copy.
def slice(offset, length)
  end

# @returns [String] A binary string starting at offset, length bytes.
def to_str(offset, length)
  end

# Copy the specified string into the buffer at the given offset.
def copy(string, offset)
end

# Compare two buffers.
def <=>$= (other)
end

include Comparable

# Resize the buffer, preserving the given length (if non-zero).
def resize($size, preserve = 0)
end

# Clear the buffer to the specified value.
def clear($value = 0, offset = 0, length = $size - offset)
end

# Data Types:
# Lower case: little endian.
# Upper case: big endian (network endian).
#
# :U8 | unsigned 8-bit integer.
# :S8 | signed 8-bit integer.
#
# :u16, :U16 | unsigned 16-bit integer.
# :s16, :S16 | signed 16-bit integer.
#
# :u32, :U32 | unsigned 32-bit integer.
# :s32, :S32 | signed 32-bit integer.
#
# :u64, :U64 | unsigned 64-bit integer.
# :s64, :S64 | signed 64-bit integer.
#
# :f32, :F32 | 32-bit floating point number.
# :f64, :F64 | 64-bit floating point number.

# Get the given data type at the specified offset.
def get(type, offset)
end

# Set the given value as the specified data type at the specified offset.
def set(type, offset, value)
end

The C interface provides a few convenient methods for accessing the underlying data buffer:

void rb_io_buffer_get_mutable(VALUE self, void **base, size_t *size);
void rb_io_buffer_get_immutable(VALUE self, const void **base, size_t *size);

In the fiber scheduler, it is used like this:

VALUE
rb_fiber_scheduler_io_read_memory(VALUE scheduler, VALUE io, void *base, size_t size, size_t length)
{
  VALUE buffer = rb_io_buffer_new(base, size, RB_IO_BUFFER_LOCKED);

  VALUE result = rb_fiber_scheduler_io_read(scheduler, io, buffer, length);

  rb_io_buffer_free(buffer);

  return result;
}

This function is invoked from io.c at various places to fill the buffer. We specifically the (base, size) tuple, along with length which is the minimum length required and assists with efficient non-blocking implementation.
The uring.c implementation in the event gem uses this interface like so:

```c
VALUE Event_Backend_URing_io_read(VALUE self, VALUE fiber, VALUE io, VALUE buffer, VALUE _length)
{
  struct Event_Backend_URing *data = NULL;
  TypedData_Get_Struct(self, struct Event_Backend_URing, &Event_Backend_URing_Type, data);

  int descriptor = RB_NUM2INT(rb_funcall(io, id_fileno, 0));

  void *base;
  size_t size;
  rb_io_buffer_get_mutable(buffer, &base, &size);

  size_t offset = 0;
  size_t length = NUM2SIZET(_length);

  while (length > 0) {
    size_t maximum_size = size - offset;
    int result = io_read(data, fiber, descriptor, (char*)base+offset, maximum_size);

    if (result == 0) {
      break;
    } else if (result > 0) {
      offset += result;
      if ((size_t)result > length) break;
      length -= result;
    } else if (-result == EAGAIN || -result == EWOULDBLOCK) {
      Event_Backend_URing_io_wait(self, fiber, io, RB_INT2NUM(READABLE));
    } else {
      rb_syserr_fail(-result, strerror(-result));
    }
  }

  return SIZET2NUM(offset);
}
```

### Buffer Allocation

The Linux kernel provides some advanced mechanisms for registering buffers for asynchronous I/O to reduce per-operation overhead.

The io_uring_register() system call registers user buffers or files for use in an io_uring(7) instance referenced by fd. Registering files or user buffers allows the kernel to take long term references to internal data structures or create long term mappings of application memory, greatly reducing per-I/O overhead.

With appropriate support, we can use IORING_OP_PROVIDE_BUFFERS to efficiently manage buffers in applications which are dealing with lots of sockets. See https://lore.kernel.org/io-uring/20200228203053.25023-1-axboe@kernel.dk/T/ for more details about how it works. I'm still exploring the performance implications of this, but the proposed implementation provides sufficient meta-data for us to explore this in real world schedulers.

PR: [https://github.com/ruby/ruby/pull/4621](https://github.com/ruby/ruby/pull/4621)

### History

**#1 - 07/03/2021 07:24 AM - ioquatix (Samuel Williams)**

This also relates to [https://bugs.ruby-lang.org/issues/13166](https://bugs.ruby-lang.org/issues/13166)

**#2 - 07/03/2021 08:08 AM - ioquatix (Samuel Williams)**

Here is the initial proposed implementation / interface: [https://github.com/ruby/ruby/pull/4621](https://github.com/ruby/ruby/pull/4621)

**#3 - 07/07/2021 08:25 PM - ioquatix (Samuel Williams)**

Okay, I have reverted the changes to try and support IO#read and IO#write. It's too complicated to implement it right now. However, I discussed it with usa (Usaku NAKAMURA) who had some ideas regarding how we can improve IO.
The current implementation is adequate for the fiber scheduler backend.

#4 - 07/08/2021 10:39 AM - Eregon (Benoit Daloze)

Does it need to be core, or could it be behind a require like require 'io/buffer'? The reason I'm asking is the C code could be reused if it's an extension (behind a require) in TruffleRuby, but not if it is core (to avoid loading C extensions during VM startup).

The IO::Buffer API looks very similar to FFI::Pointer, to the point it feels redundant with it. Maybe a FFI::Pointer subclass could be used if some extra methods/state is needed. Turned another way: what would be possible with IO::Buffer that is not possible with FFI::Pointer?

It might be more valuable to make ffi a default or bundled gem, which also brings much more capabilities. ffi is already a bundled gem for both JRuby and TruffleRuby.

#5 - 07/08/2021 10:55 AM - ioquatix (Samuel Williams)

Eregon (Benoit Daloze) thanks for your discussion.

Something like this is required for the fiber scheduler interface. It's also required for efficient IO. Many people have asked for this feature, maybe there is something I don't know but why they didn't use FFI::Pointer and why is there interest in IO::Buffer from other people? If they could already use FFI::Pointer, why didn't they?

I'm not against FFI::Pointer but there are probably some subtle differences in that I'm initially interested in the IO layer and zero-copy IO. I'm not sure how efficiently FFI::Pointer is implemented either, but this will be something we can map directly to our use case which is specifically IO related. Network IO does have specific requirements around efficient decode of binary data.

It might be more valuable to make ffi a default or bundled gem, which also brings much more capabilities. ffi is already a bundled gem for both JRuby and TruffleRuby.

This may be a problem as the fiber scheduler is part of the core interface. So, whatever we have, it must be part of Ruby core? I'm pretty keen to keep the definition of the fiber scheduler as simple as possible, so introducing a relatively straightforward memory buffer is probably preferable to pulling in all of ffi, at least from a complexity PoV.

#6 - 07/08/2021 03:29 PM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-5:

why is there interest in IO::Buffer from other people?

I was not aware of that, did people specifically ask for IO::Buffer?

I'm not sure how efficiently FFI::Pointer is implemented either.

It's very efficient, most likely as efficient or better than IO::Buffer. It's already intensified on CRuby, TruffleRuby and JRuby. Doing that work again for IO::Buffer feels redundant to me.

but this will be something we can map directly to our use case which is specifically IO related.

You can build a FFI::Pointer around a raw address, so it's also possible to ensure it's aligned, etc.

Network IO does have specific requirements around efficient decode of binary data.

FFI::Pointer has read/write/get/put_byte/short/int/long, etc, so I think that should cover it.

This may be a problem as the fiber scheduler is part of the core interface. So, whatever we have, it must be part of Ruby core?

Not necessarily, nothing forces the scheduler interface to yield an IO::Buffer object, it could let the user choose whatever they want to use to read/write to a given native address. I think that's actually more flexible, and might avoid an extra IO::Buffer allocation which would otherwise not be needed.

#7 - 07/09/2021 06:47 AM - ioquatix (Samuel Williams)

Does FFI::Pointer have locking mechanism suitable for IO? Does it have a mutability model suitable for reading and writing? Does it allocate page aligned mapped memory suitable for zero-copy IO?

Even if I agree that it was suitable, how can we use it since it's not part of core Ruby, so there is no way it can be used in io.c or scheduler.c.
Okay, the PR is ready for review: [https://github.com/ruby/ruby/pull/4621](https://github.com/ruby/ruby/pull/4621)

Here is how it's used:

- `uring.c`: [https://github.com/socketry/event/blob/b40bb0b174aed4cc3fed0f0eaafdd73f2a6a6f4c/ext/event/backend/uring.c#L265-L365](https://github.com/socketry/event/blob/b40bb0b174aed4cc3fed0f0eaafdd73f2a6a6f4c/ext/event/backend/uring.c#L265-L365)
- `epoll.c`: [https://github.com/socketry/event/blob/b40bb0b174aed4cc3fed0f0eaafdd73f2a6a6f4c/ext/event/backend/epoll.c#L269-L414](https://github.com/socketry/event/blob/b40bb0b174aed4cc3fed0f0eaafdd73f2a6a6f4c/ext/event/backend/epoll.c#L269-L414)
- `kqueue.c` implementation largely the same as `epoll`.
- `select.rb`: [https://github.com/socketry/event/blob/b40bb0b174aed4cc3fed0f0eaafdd73f2a6a6f4c/lib/event/backend/select.rb#L56-L101](https://github.com/socketry/event/blob/b40bb0b174aed4cc3fed0f0eaafdd73f2a6a6f4c/lib/event/backend/select.rb#L56-L101)

In `io_uring` implementation, the data buffer is passed directly to the OS for zero-copy I/O.

A brief overview of the implementation:

- It provides a fast path from internal IO buffering to the fiber scheduler.
- It's primarily an object that represents a `(void*, size_t)` tuple.
- It can allocate it's own memory using malloc, mmap or VirtualAlloc (mainly for testing).
- It can also map File objects into memory (experimental).
- It provides some basic provisions for getting and setting data.
- It provides a locking mechanism to prevent incorrect usage while the buffer is being used by the OS/system.
- It provides mutable/immutable flag to validate correct usage when reading/writing.

Going forward, I would like to see a more elaborate model where we can read and write directly using these buffers. We want a fast path for binary protocols like DNS, HTTP/2 etc. This implementation of get/set is 4x faster than String#unpack in my limited testing.

**#9 - 07/09/2021 07:26 AM - ioquatix (Samuel Williams)**
- Description updated

**#10 - 07/09/2021 07:44 AM - ioquatix (Samuel Williams)**
- Description updated

**#11 - 07/10/2021 03:30 AM - ioquatix (Samuel Williams)**
- Description updated

Add notes about buffer allocation.

**#12 - 07/13/2021 10:35 AM - Eregon (Benoit Daloze)**

API-wise: integer flags feel not so Ruby-like. How about symbols instead? Or are those flags only meant to be used from C?

```ruby
def to_str(offset, length)
  #def to_str(offset, length) seems problematic, the coercion protocol is to_str() (no arguments). So it should be another method name, or also work if no arguments are given. I don't think in general an IO::Buffer should be considered implicitly a String, so probably best to not have to_str at all.
  #to_s(offset = 0, length = size) seems better anyway, to_s is for explicit conversions which is the point of that method.
``` 

**#13 - 07/14/2021 12:31 AM - ioquatix (Samuel Williams)**

Eregon (Benoit Daloze) thanks for the feedback.

The flags are more efficient and for the current design they are mostly implementation specific. I'm not sure how you implement multiple flags with symbols? For the initial design we can actually avoid exposing any flags to Ruby - it might make sense to cut down the interface to just the most basic public interface required to implement the scheduler hooks.

```ruby
def to_str(offset, length)
  #def to_str(offset, length)

  #Yes, totally agree, we can change this to #string(offset, length) which makes sense.
```

**#14 - 09/02/2021 03:31 PM - dsisnero (Dominic Sisneros)**

this is what memoryview is supposed to give us. I agree some IO class is needed. This class should implement memoryview interface and IO class should have a read_into method and write_into method that takes a memoryview compatible object.

see #17834
see #17833
see #17832

**#15 - 09/02/2021 03:43 PM - dsisnero (Dominic Sisneros)**

I also propose FFI::Pointer and Fiddle::Pointer and Fiddle::Memoryview be combined into one library and made available to all IO. When we get IO and memoryview compatible objects we are going to want to use those objects in ruby and having two similar libraries that people only think is for c extensions will discourage use.
dsisnero (Dominic Sisneros) I'm not convinced we can implement the required semantics with the existing implementations.

bytestring = Bytes.new('this is a string')

Looking at this usage, this is already a bad model for IO. We need fixed size buffers with constraints around mutability. We need guarantees on memory mapped allocations and alignment. We need semantics which allow for efficient reading and writing. Anything that requires copying memory is a non-starter.

I also looked at the memory view implementation. It's very complicated which put me off a bit. I prefer more simple design.

That being said, I have no problem with augmenting IO::Buffer to map into a memory view interface.

#17 - 09/04/2021 11:44 PM - ioquatix (Samuel Williams)

IO class should have a read_into method and write_into

This is very hard to implement, e.g. OpenSSL::SSL::Socket.

#18 - 09/14/2021 10:56 PM - dsisnero (Dominic Sisneros)

+1 that being said, I have no problem with augmenting IO::Buffer to map into a memory view interface.

f = File.open(FILENAME, 'rb')
bytearray = ByteArray.new(File.size(FILENAME))
# ByteArray implements memoryview and is mutable  ByteString implements memoryview and is immutable
f.readinto(bytearray)

this gives

guarantees on memory mapped allocations and alignment. We need semantics which allow for efficient reading and writing

#19 - 09/27/2021 06:16 AM - ioquatix (Samuel Williams)

dsisnero (Dominic Sisneros) would you have time sometime this week to have a quick face to face chat? It would be good to discuss the proposal with you and figure out a consistent and cohesive way forward. Based on what I can see, it seems like you have a lot of experience in this area.

#20 - 09/27/2021 06:40 AM - ioquatix (Samuel Williams)

Eregon (Benoit Daloze) I revisited this code.

def to_str(offset, length)

I'm not sure if there is any problem making an IO buffer implicitly convertible to a string. It means you can send a buffer to a function that takes a string, and it will implicitly convert it to the full buffer.

#21 - 09/27/2021 09:43 AM - Eregon (Benoit Daloze)

ioquatix (Samuel Williams) wrote in #note-20:

Eregon (Benoit Daloze) I revisited this code.

def to_str(offset, length)

I'm not sure if there is any problem making an IO buffer implicitly convertible to a string. It means you can send a buffer to a function that takes a string, and it will implicitly convert it to the full buffer.

Should it take no arguments then?
AFAIK implicit conversion methods never takes arguments.
I'm not sure if it's a good idea to make a full copy of the bytes implicit.

#22 - 10/19/2021 05:11 AM - ko1 (Koichi Sasada)

Today we read the ticket (not all comments, sorry) and mame, ko1 has comment:

mame: I cannot understand what is finally needed. Doesn't String with ASCII-8BIT work?
ko1: I can't understand how to use it with IO? Only for scheduler?
mame: I cannot understand what is finally needed. Doesn't String with ASCII-8BIT work?

String is both insufficient and inefficient. You can check how read and write on strings work with internal frozen copies, for example, it's both performance and semantic issue. We can't expose fake string to scheduler which might be one other option, but it has a lot of edge cases. I tried it already.

ko1: I can't understand how to use it with IO? Only for scheduler?

Several examples are given in PR, including new scheduler hooks and tests. Initially only for scheduler, but I believe we should expose to application code.

Similar concepts exists in Crystal: https://crystal-lang.org/api/1.1.1/IO/Memory.html & https://crystal-lang.org/api/1.1.1/Bytes.html but this is designed more for high performance I/O.

In the description's code, there is lock and unlock. Are those supposed to be thread-safe? If yes I think you'd need to synchronize in almost every method, if it's possible to access the buffer without GVL.

I think it's better to only allow "lock" on creation, to there is no dynamic lock or unlock, which makes everything more complex.

In fact, do we even need resizable buffers? IMHO using another buffer seems much cleaner if one needs to grow it.

The new interface feels really big and hard to understand it as a whole. IMHO the thread-unsafe parts (this and that) should be removed, and the interface simplified as much as possible, and then it would be a lot easier to review.

e.g., if it's a fixed-size buffer then it's already much easier to reason about than some IO::Buffer doing everything.

In the description's code, there is lock and unlock. Are those supposed to be thread-safe? If yes I think you'd need to synchronize in almost every method, if it's possible to access the buffer without GVL.

No, instances of this class should not be shared between threads. However, there would be some cases where this might be okay, e.g. if the buffer is immutable. We definitely want to avoid any kind of synchronisation overheads for performance reasons.

I think it's better to only allow "lock" on creation, to there is no dynamic lock or unlock, which makes everything more complex.

A buffer can be used across multiple I/O operations necessitating locking and unlocking, not unlike the already existing implementation on String. The reason for this complexity is to prevent user error and to model the fact that the OS can use a buffer which may be for a duration of time outside of the GVL and it should not be changed while in use by the OS.

Additionally, I'd argue that String implementation is way more complex and poorly understood/documented, and so far we seem happy for that? There are definitely some very odd edge cases when using Strings as buffers, some of which I already reported and fixed as potential security issues, both in CRuby and one of JRuby/TruffleRuby.

In fact, do we even need resizable buffers? IMHO using another buffer seems much cleaner if one needs to grow it.

Celluloid introduced fixed size buffers and it was very hard to use correctly. In the end, I couldn't even use it in async because it was so impractical. So yes, resizable buffers are absolutely needed and can be efficiently implemented, either by memory mapping or copying in the worst case. If you don't implement this, the user ends up having to do it by hand = more bugs & less performance.

The new interface feels really big and hard to understand it as a whole.

Based on my experience of io.c and string.c I completely disagree with this assertion. I feel that this is a far simpler, well abstracted, isolated, robust interface for dealing with binary data in conjunction with IO in comparison to what we already have. I literally spent like a month trying to retrofit the aforementioned code but it was like a house of cards. Move one thing and the entire thing collapses. Unfortunately the Ruby IO & String class is really overloaded and has become a significant burden to implementing predictable, efficient and robust network protocols.

The full implementation is given, including the example usage, tests, and also the implementation of the scheduler hooks. Additionally, a full consumer implementation in the Event gem is given: https://github.com/socketry/event/blob/master/ext/event/selector/uring.c#L359-L459 (there are 03/20/2022 8/10
also implementations for pure Ruby [select], kqueue, and epoll).

If you think there is something wrong with this implementation or that it can be greatly simplified, please propose specific changes to the implementation that achieve this and still maintain safety, efficiency, performance, etc. I welcome any such changes and would be most grateful for your insights on such improvements. You've got a full end-to-end PR to work with, there is nothing missing or theoretical here.

#27 - 10/29/2021 01:26 AM - ioquatix (Samuel Williams)

matz (Yukihiro Matsumoto) you said you are positive on this feature. Can you confirm that we can merge this PR? Even we can mark it as experimental, but it would be great to start testing with 3.1 preview release.

akr (Akira Tanaka) i believe we addressed your concerns preventing modifying buffer while it's in use by OS by preventing Ruby from calling #unlock. Do you have any other concerns? Even if you can't enumerate it all now, we can try with Ruby 3.1 preview release and address any further concerns over the next few months.

Thanks everyone.

#28 - 11/08/2021 02:35 AM - matz (Yukihiro Matsumoto)

I am not fully satisfied with the quality of the code (at the last time I checked a while ago), but basically, I agree with the merging. So let us experiment with it.

Matz.

#29 - 11/08/2021 09:27 PM - ioquatix (Samuel Williams)

matz (Yukihiro Matsumoto) thanks, I will rebase and merge it with experimental warning.

#30 - 11/10/2021 06:24 AM - ioquatix (Samuel Williams)

I have merged this. We will follow up with additional changes in new tickets as needed.

#31 - 11/10/2021 09:27 PM - mame (Yusuke Endoh)

The change caused SEGV on Solaris.


Thread B (Thread 84 {LWP 84}):

```
# 0 0xfef2054c in __systemcall16 () from /lib/libc.so.1
# 1 0xfef05d18 in __lwp_sigmask () from /lib/libc.so.1
# 2 0xef0f6d4 in call_user_handler () from /lib/libc.so.1
# 3 <signal handler called>
# 4 0xefef94 in __waitid () from /lib/libc.so.1
# 5 0xfeebfa84 in __waitpid () from /lib/libc.so.1
# 6 0xef0e920 in waitpid () from /lib/libc.so.1
# 7 0xef0f163c in system () from /lib/libc.so.1
# 8 0x0025f854 in rb_vmbugreport (ctx=ctx@entry=0x5f18880) at vm_dump.c:1016
# 10 0x001ae5dc in sigsegv (sig=11, info=0x5f18b38, ctx=0x5f18880) at signal.c:964
```

03/20/2022
The commit not only introduces IO::Buffer but also includes many changes against IO internal. It heavily uses rb_io_t * instead of a file descriptor, but accessing rb_io_t * without GVL requires special care, and IO internal heavily uses "nogvl" code.

---

#32 - 11/10/2021 09:04 AM - ioquatix (Samuel Williams)

Thanks mame (Yusuke Endoh), yes this change adds fiber scheduler hooks for low level file read/write operations and this necessitates passing the IO object around rather than just the file descriptor integer. The changes are mostly cosmetic though and the accesses to file descriptor should only occur in the same context where it was valid previously, it was mostly a mechanical change to pass the rb_io_t * rather than raw file descriptor. In any case, I'll check what is the problem.

#33 - 11/10/2021 12:23 PM - ioquatix (Samuel Williams)

My initial assessment, based on the changes we made, is that we are potentially incorrectly using fd before this PR is applied. For it to fail in the way it is, it means we are still using the previous value of fptr->fd even though it was already set to -1 and/or closed.

This PR may help work around the issue: https://github.com/ruby/ruby/pull/5100